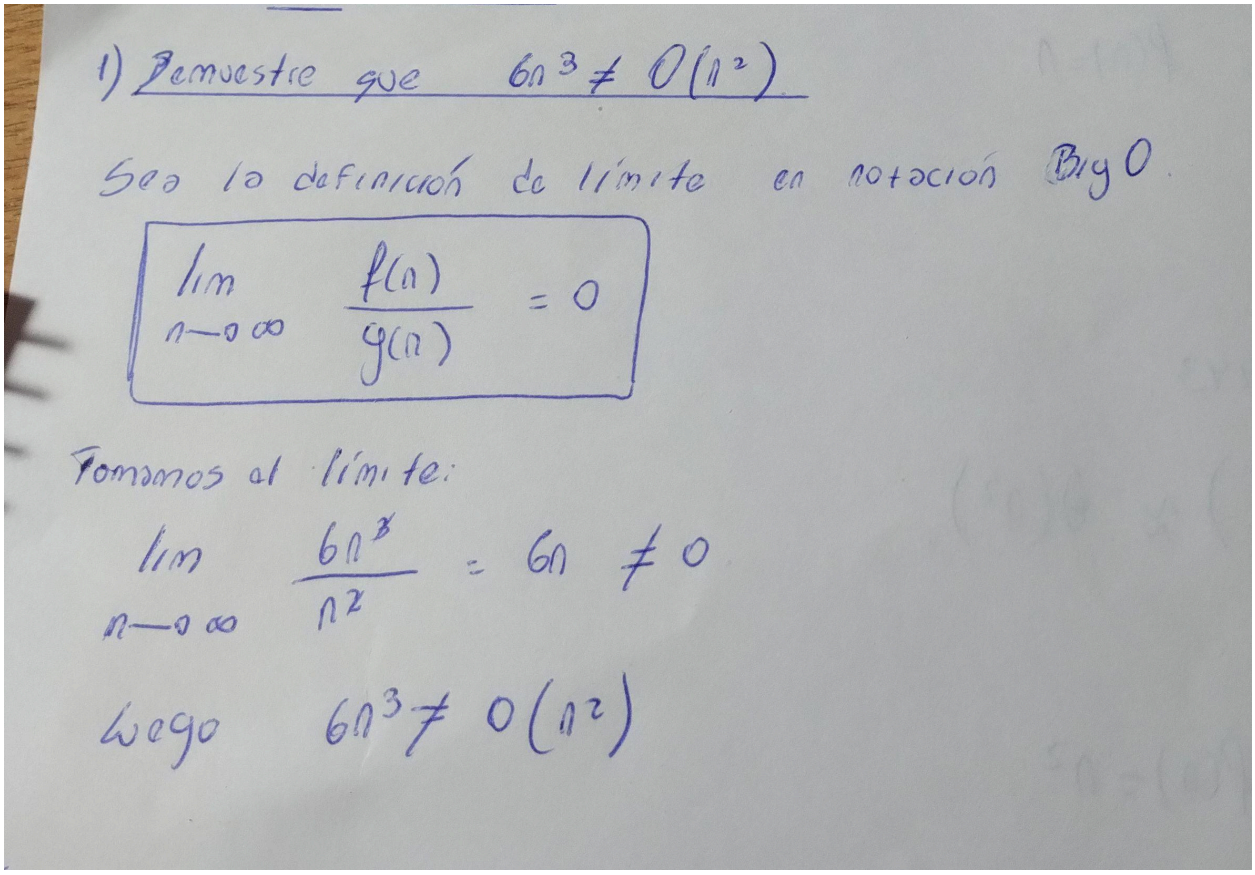
Ejercicio 1:

Demuestre que $6n^3 \neq \boldsymbol{O}(n^2)$.



Ejercicio 2:

¿Cómo sería un array de números (mínimo 10 elementos) para el mejor caso de la estrategia de ordenación Quicksort(n) ?

En el mejor caso para el algoritmo Quicksort, el arreglo estaría completamente balanceado en cada partición. Esto significa que en cada paso de partición, dividiríamos el arreglo en dos partes casi iguales.

[1,2,3,4,6,5,7,8,9,10]

Ejercicio 3:

Cuál es el tiempo de ejecución de la estrategia **Quicksort(A), Insertion-Sort(A)** y **Merge-Sort(A)** cuando todos los elementos del array A tienen el mismo valor?

- **Quicksort:** O(n²)
- **Insertion-Sort:** O(n)
- **Merge-Sort:** O(nlogn)

Ejercicio 4:

Implementar un algoritmo que ordene una lista de elementos donde siempre el elemento del medio de la lista contiene antes que él en la lista la mitad de los elementos menores que él. Explique la estrategia de ordenación utilizada.

Ejemplo de lista de salida

7	3	2	8	5	4	1	6	10	9

```python
#Ejercicio 4
def sortList(L):

    L.sort()
    print()
    L[2], L[len(L) // 2] = L[len(L) // 2], L[2]
    L[0], L[len(L) - 1] = L[len(L) - 1], L[0]

    return L
```

Ejercicio 5:

Implementar un algoritmo **Contiene-Suma(A,n)** que recibe una lista de enteros A y un entero n y devuelve True si existen en A un par de elementos que sumados den n. Analice el costo computacional.

1. Ordenar el arreglo (utilizar Merge-Sort).
2. Ir al final y al principio de la lista. Utilizar dos punteros (i y j) respectivamente.
3. Sumar los dos elementos: A[i] + A [j]
4. Si la suma es menor a “n” mover el puntero i a la derecha. Si la suma es mayor a “n”, mover el puntero “j” a la izquierda.
5. Volver al paso 2
6. Repetir hasta encontrar la suma o bien hasta que llegar a que i ==j (esto significa que el array no contiene la suma “n”)

Implementación en Python:

```python
#Ejercicio 5
def contieneSuma(A,n):
    i = 0
    j = len(A) - 1
    while i != j:
        suma = A[i] + A[j]
        if suma == n:
            return True

        if suma < n:
            i += 1
        else:
            j -= 1

    return False
```

Ejercicio 6:

Investigar otro algoritmo de ordenamiento como BucketSort, HeapSort o RadixSort, brindando un ejemplo que explique su funcionamiento en un caso promedio. Mencionar su orden y explicar sus casos promedio, mejor y peor.

Algoritmo de Shell-Sort:

El método de ordenamiento Shell consiste en dividir el arreglo (o lista de elementos) en intervalos (o bloques) de varios elementos para organizarlos después por medio del ordenamiento de inserción directa.

Funcionamiento:

1. Definir el intervalo inicial (gap): Se elige un intervalo inicial, comúnmente la mitad del tamaño del arreglo, pero esto puede variar según la implementación.
2. Dividir el arreglo en subarreglos: Se dividen los elementos del arreglo en subarreglos separados por el intervalo definido.
3. Aplicar algoritmo de inserción para cada subarreglo: Se ordena cada subarreglo usando el algoritmo de inserción. Dado que los elementos de los subarreglos están más cerca entre sí que en el arreglo original, el algoritmo de inserción es más eficiente en este contexto.
4. Reducir el intervalo (gap): Se reduce el intervalo, lo que significa que los subarreglos serán más pequeños en la próxima iteración.
5. Repetir el proceso hasta que el intervalo sea 1: Se repiten los pasos 2 a 4, reduciendo gradualmente el intervalo hasta que se alcanza el valor 1.

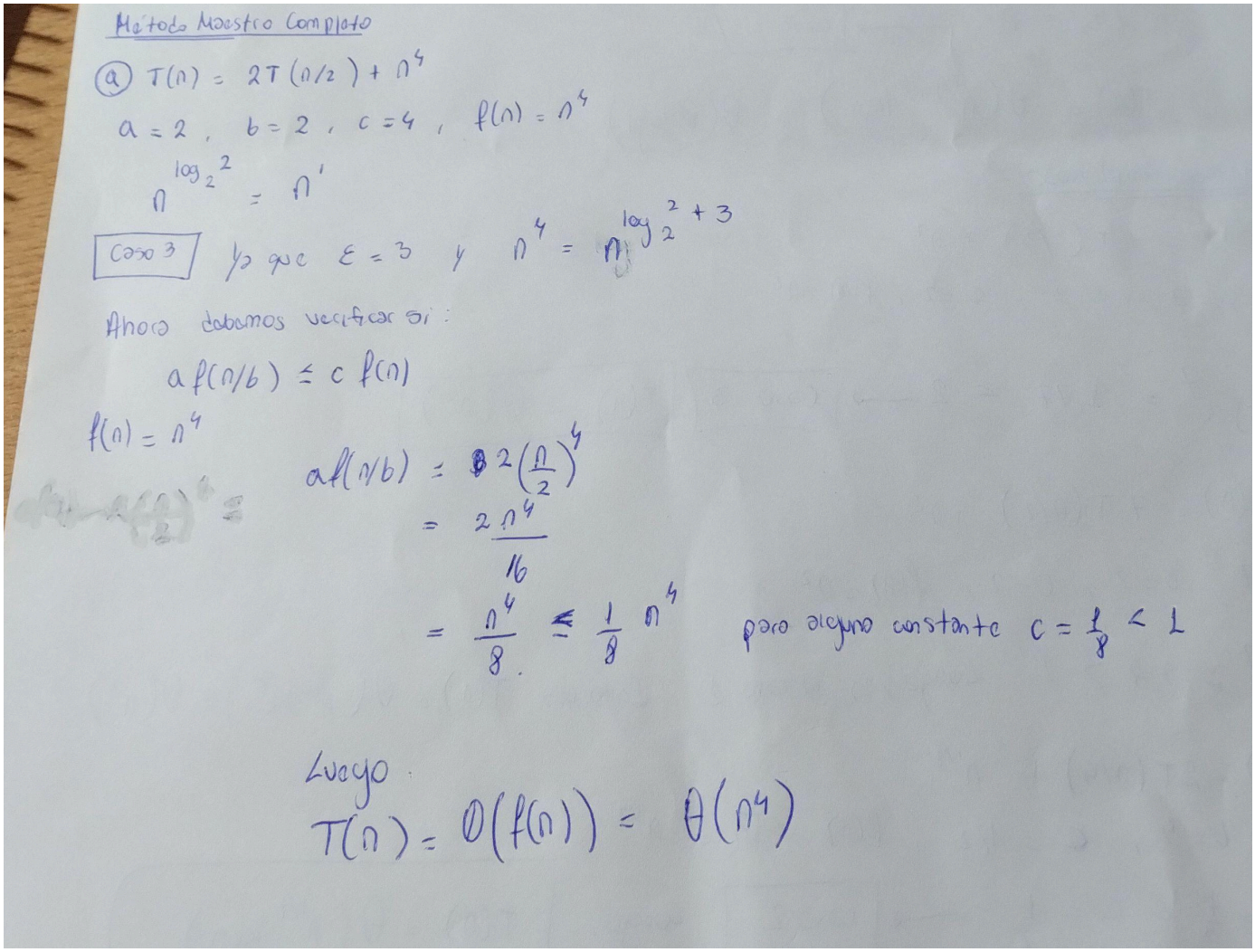
Complejidad:

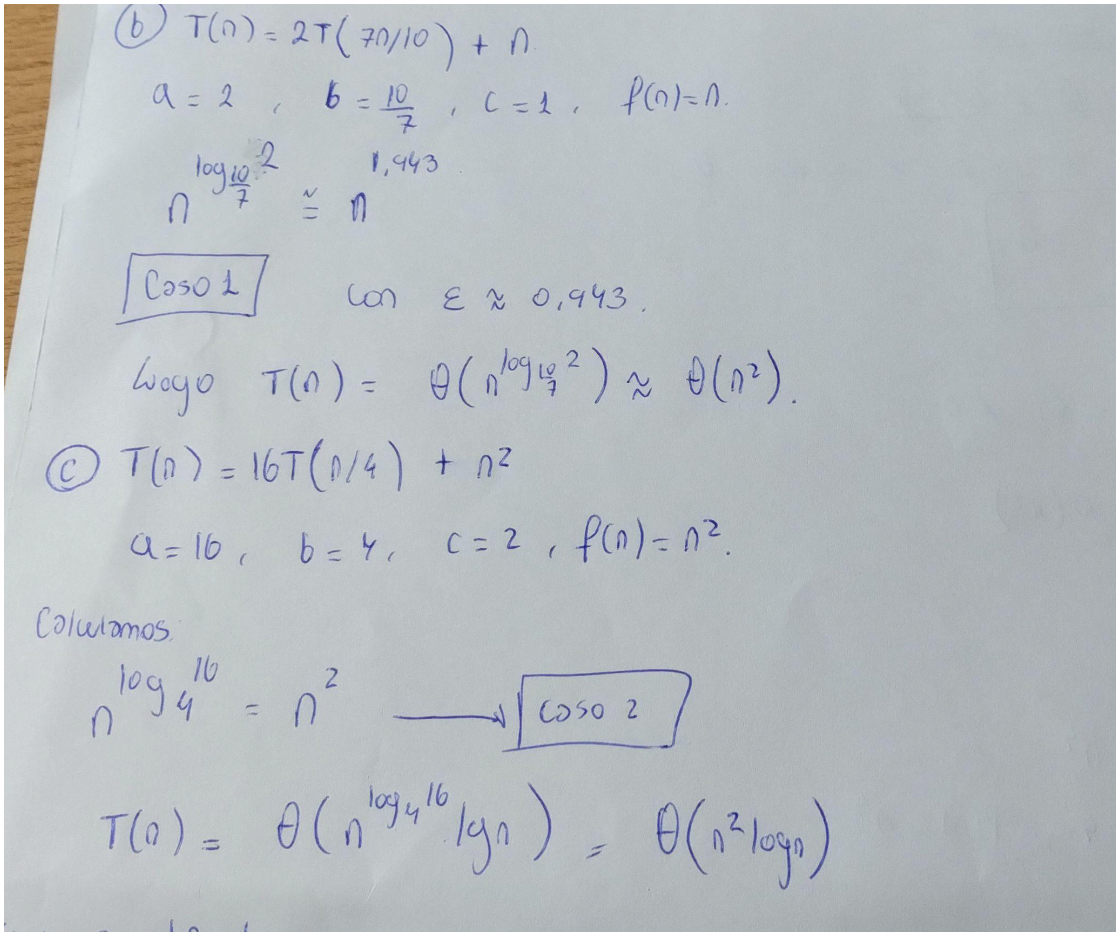
- **Caso promedio:** El tiempo de ejecución en el caso promedio suele ser algo mejor que O(n^2) pero peor que O(n log n).
- **Mejor caso:** Puede aproximarse a O(nlogn)
- **Peor caso:** $O(n^2)$

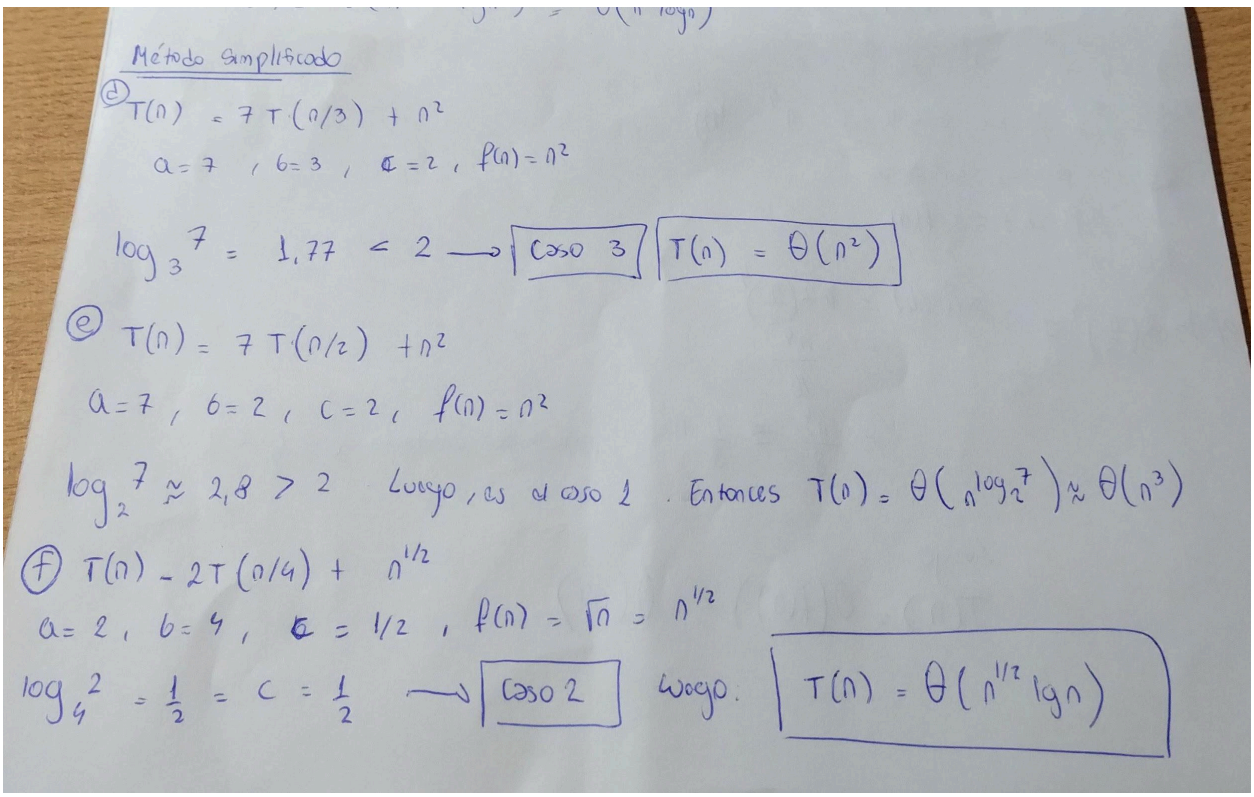
Ejercicio 7:

A partir de las siguientes ecuaciones de recurrencia, encontrar la complejidad expresada en Θ(n) y ordenarlas de forma ascendente respecto a la velocidad de crecimiento. Asumiendo que T(n) es constante para n <= 2. Resolver 3 de ellas con el método maestro completo: T(n) = a T(n/b) + f(n) y otros 3 con el método maestro simplificado: T(n) = a T(n/b) + n^c

a. T(n) = 2T(n/2) + n^4
b. T(n) = 2T(7n/10) + n
c. T(n) = 16T(n/4) + n^2
d. T(n) = 7T(n/3) + n^2
e. T(n) = 7T(n/2) + n^2
f. T(n) = 2T(n/4) + √n




(b) $T(n) = 2T(7n/10) + n$
$a = 2$, $b = \frac{10}{7}$, $c = 1$, $f(n) = n$.
$n^{\log_{\frac{10}{7}} 2} \approx n^{1,943}$
Caso 1 con $\varepsilon \approx 0,943$.
Luego $T(n) = \theta(n^{\log_{\frac{10}{7}} 2}) \approx \theta(n^2)$.
(c) $T(n) = 16T(n/4) + n^2$
$a = 16$, $b = 4$, $c = 2$, $f(n) = n^2$.
Calculamos
$n^{\log_4 16} = n^2$ → Caso 2
$T(n) = \theta(n^{\log_4 16} \lg n) = \theta(n^2 \log n)$


Método simplificado
(d) $T(n) = 7T(n/3) + n^2$
$a = 7$, $b = 3$, $c = 2$, $f(n) = n^2$
$\log_3 7 = 1,77 < 2$ → Caso 3 $T(n) = \theta(n^2)$
(e) $T(n) = 7T(n/2) + n^2$
$a = 7$, $b = 2$, $c = 2$, $f(n) = n^2$
$\log_2 7 \approx 2,8 > 2$ Luego, es el caso 1. Entonces $T(n) = \theta(n^{\log_2 7}) \approx \theta(n^3)$
(f) $T(n) = 2T(n/4) + n^{1/2}$
$a = 2$, $b = 4$, $c = 1/2$, $f(n) = \sqrt{n} = n^{1/2}$
$\log_4 2 = \frac{1}{2} = c = \frac{1}{2}$ → Caso 2 Luego. $T(n) = \theta(n^{1/2} \lg n)$

A tener en cuenta:

1. **Usen lápiz y papel primero**
2. ~~**No se puede utilizar otra Biblioteca mas alla de algo1.py y linkedlist.py**~~
3. **Hacer una análisis por cada algoritmo implementado del caso mejor, el caso peor y una perspectiva del caso promedio.**